GD2S02 – Software Engineering for Games

Test Driven Development

Overview

- Test Driven Development
- Why TDD?
- Refactoring

Test Driven Development (TDD)

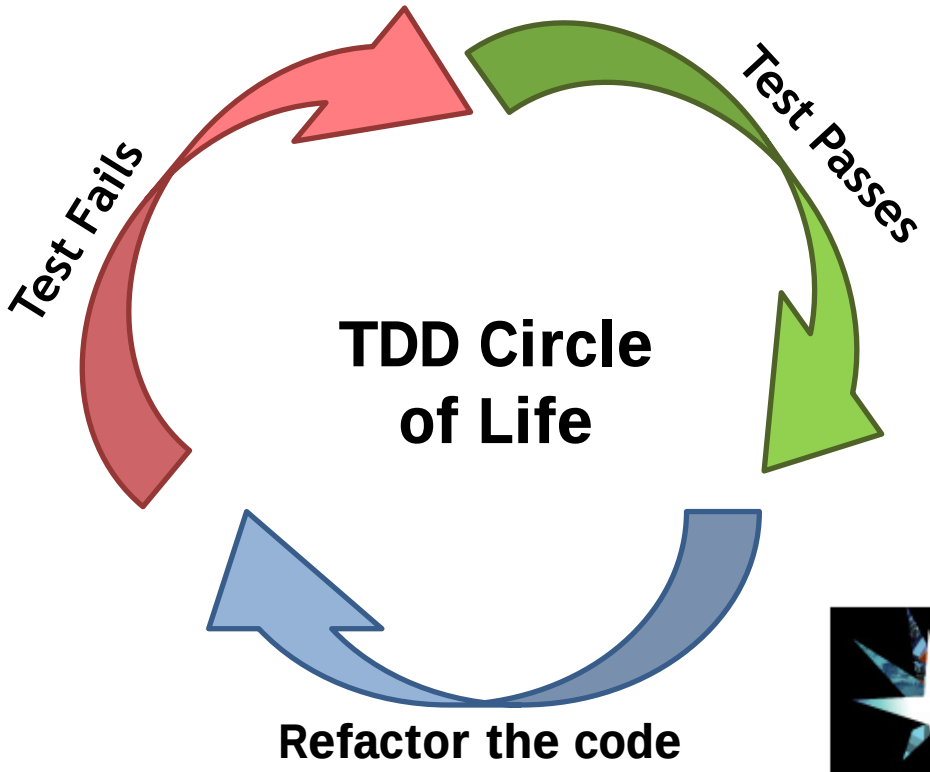
- "TDD is a a software development process that relies on the repetition of a very short development cycle:
  - First the developer writes an (initially failing) automated test case that defines a desired improvement or new function.
  - Then produces the minimum amount of code to pass that test.
  - Refactors the new code to acceptable standards."

Test Driven Development (TDD)

- Test-driven development is related to the test-first programming concepts of extreme programming.
- The Wikipedia definition of "**Extreme programming** (**XP**) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoint[illegible]" which new customer requirements can be adopte[illegible]

TDD Circle of Life

- In test driven development, coded unit tests are written first to fail before the functionality is implemented.
- Red/Green/Refactor



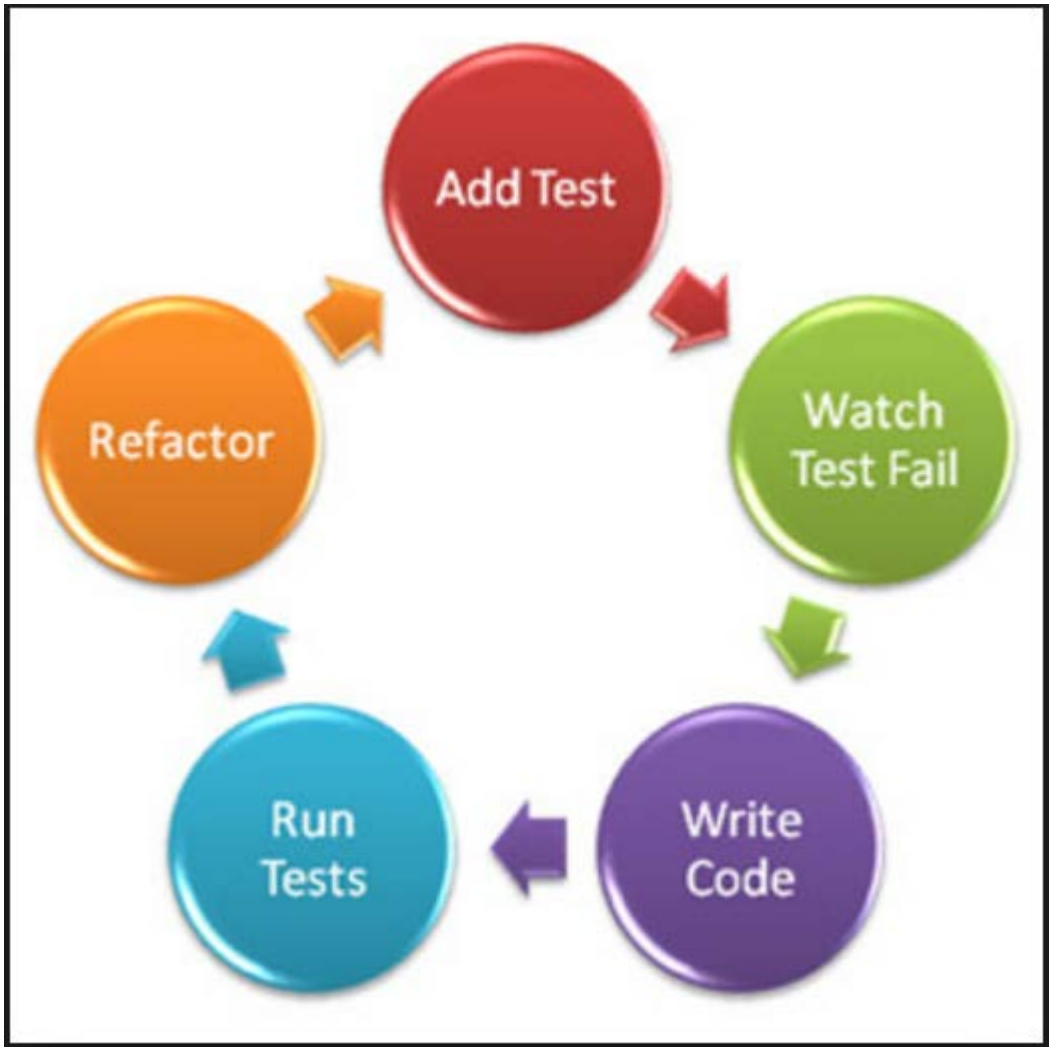
How TDD technique works?

- **Understand the requirements of the story that you are working on**
- **Red: Create a test and make it fail**
  - Show the intent of the new code
  - Write the test as if the code already existed.
  - Think critically about the design.

How TDD technique works?

- **Green:** Do whatever it takes to make the test pass.
  - If you see the full implementation, add the new code.
  - If you don't, do just enough to get the test to pass.
  - Keep it simple.
- **Refactor:** Go back and clean up any code or sins committed while trying to get the test to pass.
  - Removing duplication
  - Make sure everything is lean, mean, and as clear as possible.

TDD Detailed Circle of Life



MEDIA DESIGN SCHOOL GAME DEV

TDD Steps

1. Add a test
   - For each new feature you want to add to your software, write a test.
   - The test should define the new function or the improvement that this feature is trying to accomplish.
   - To write the test, the developer needs to understand the feature's specification and requirements clearly.
   - Write the test as if the code already exists.

TDD Steps

2. Run tests and see if the new test will fail/pass
   - To validate the test harness.
   - To show that the new test requires new code to support it.
   - To rule out the possibility that the new test is flawed and will always pass even with no code change.

TDD Steps

3. If the new test fails, write some code to make it pass.
   - Write the minimal code to make the test case pass.
   - The added code is not perfect and will need to be reviewed.
   - Do not write code beyond the functionality checked by the test case.
   - Keep it simple.

TDD Steps

4. Run the tests again
   - Make sure that the added code met the requirements.
   - Make sure that the added code did not break any other test cases/features.

TDD Steps

5. Refactor code

- Review the newly added code to make sure that:
  - The code is moved to where it logically belongs.
  - No duplications.
  - The naming conventions and coding standards are followed.
  - The code is still readable and maintainable.

TDD Steps

6. Repeat
   - Add a new test representing a new feature.
   - Repeat the same steps.

Rules of Thumb

- Rule #1: *Don't write any new code until you first have a failing test.*
  - It might not be possible to follow the rules 100 percent; however, follow the spirit!
  - Do not write any more code than absolutely necessary!
  - Think about the value of what you are adding!
- Rule #2: *Test everything that could "possibly" break.*
  - Key word here is **possibly**!
  - Not literally testing everything!

Tests to deal with complexity

- We face a lot of complexity while writing code
  - Classes, different types, arguments, visibility, decoupling: design decisions
  - TDD enables us to analyse these in simplified steps.
- Manifest what you need in the form of a test; once you create the code that will be only for which you need.
- Think: TDD is a design technique rather than being about testing with testing as a core part of it.

Why Test Driven Development (TDD)?

- TDD produces high quality software in less time than the older methods.
- Both the developer and the tester tries to anticipate how the application and the features will be used.
- TDD ensures that all the units of the software product have been tested for optimum functionally.
- Less time and money are spent on debugging because tests are carried out from the start of the design cycle.
- Helps to document the design.
- Encourages the design of testable code.

TDD Example

- We want to write a function to implement the factorial using TDD.
- The function output should be 1,1,2,6,24,120,……
- Think of the tests as{0,1}{1,1}{2,2}{3,6}{4,24}{5,120}..
- Start with a function called “Test Factorial”
- Check for the first input.
- When you write that the build will not work because there is no definition for factorial

```
void TestFactorial()
{

    assert(factorial(0) == 1);
}
```

TDD Example

- Add the definition of factorial and enough code to make the test pass.

```
int factorial(int n)
{
    return 1;
}
```

- Then think about the second test case

  This code will pass because

  both factorial 0 and 1 return 1

```
void TestFactorial()
{

    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
}
```

- Add another test case when n = 2

  This is going to assert because

  the factorial function will return 1.

```
void TestFactorial()
{

    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(2) == 2);
}
```

TDD Example

- Add enough code to make the test case pass

```
int factorial(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    if (n > 1) return n * 1;

}
```

- Add other test cases, test will fail

```
void TestFactorial()
{
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(2) == 2);
    assert(factorial(3) == 6);
    assert(factorial(4) == 24 );
    assert(factorial(5) == 120);
}
```

TDD Example

- Add enough code to make it pass

```
int factorial(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    if (n > 1) return n * factorial(n - 1);

}
```

- By now we have the function and the tests.

Common question from *The Agile Samurai**

- **STUDENT:** Master, how does unit testing not slow teams down? I mean, you're writing double the amount of code, aren't you?
- **MASTER:** If programming were merely typing, that would be true. The unit tests are there to confirm that as we make changes to our software, the universe still unfolds as expected. This saves us time by not having to manually regression test the entire system every time we make a change.

** [The Agile Samurai by Jonathan Rasmusson, The Pragmatic Programmers, 2010]*

Refactoring

- “A change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior”, Fowler.
- Refactoring is the practice of continuously making small, incremental design improvements to your software without changing the overall external behavior.
- It is not about adding a new functionality or fixing a bug, it is about making the code easier to understand and change.

** [The Agile Samurai by Jonathan Rasmusson, The Pragmatic Programmers, 2010]*

Refactoring

- Renaming poorly named variables/functions is part of the refactoring process.
- Refactoring improves readability and maintainability of the code.
- When you refactor, proceed in a series of small transformations

** [The Agile Samurai by Jonathan Rasmusson, The Pragmatic Programmers, 2010]*

Refactoring Tips

1. Proceed in a series of small transformations.
2. Run tests after each step, and make sure that they still pass.
3. It is easier to rearrange the code correctly if you don't simultaneously try to change its functionality.
4. It is easier to change functionality when you have clean code.
5. Refactoring implies equivalence; the beginning and end products must be functionally identical.

** [The Agile Samurai by Jonathan Rasmusson, The Pragmatic Programmers, 2010]*

Refactoring Example

```
public bool DealerWins(Hand hand1)
{
    var h1 = hand1;
    int sum1 =0;
    foreach (var c in h1)  {
            sum1 += Value(c.Value, h1);
    }
    var h2 = DealerManager.Hand;
    int sum2 =0;
    foreach (var c in h2) {
            sum2 += Value(c.Value, h2);
   }
        if (sum2>=sum1) {
            return true;
           }
           else    return false;
            return false;
}
```

```
public bool DealerWins(Hand PlayerHand)
  {
      int   PlayerHandValue = 0;
      foreach (var card in PlayerHand)  {
            PlayerHandValue +=
DetermineCardValue(card.Value, PlayerHand);
      }
      var DealerHand = DealerManager.Hand;
      int DealerHandValue =0;
      foreach (var card in DealerHand) {
            DealerHandValue +=
DetermineCardValue (card.Value,
DealerHand);
     }
return DealerHandValue >= PlayerHandValue
 }
```

Refactoring Example

```
public bool DealerWins(Hand PlayerHand)
  {
      int   PlayerHandValue = GetHandValue(PlayerHand);
      int  DealerHandValue = GetHandValue(DealerManager.Hand);
            return DealerHandValue >= PlayerHandValue
}
  private int  GetHandValue(Hand hand)
   {
            int HandValue = 0;
            foreach (var card in hand)
             {
             HandValue += DetermineCardValue(card, hand);
            }
            return HandValue;
   }
```

Refactoring Example

- Looking back at our TDD Factorial example code and trying to factorize that

```
int factorial(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    if (n > 1) return n * factorial(n - 1);

}
```

```
unsigned factorial(unsigned number)
{
    if (number == 0 || number == 1) return 1;
    if (number > 1) return number * factorial(number - 1);
}
```

```
void TestFactorial()
{
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);
    assert(factorial(2) == 2);
    assert(factorial(3) == 6);
    assert(factorial(4) == 24 );
    assert(factorial(5) == 120);
}
```

```
void TestFactorial()
{
    unsigned tests[][2] = { {0,1},{1,1},{2,2},{3,6},{4,24},{5,120} };
    unsigned test_count = sizeof(tests) / sizeof(tests[0]);
    for (unsigned i = 0; i < test_count; ++i)
    {
        assert(factorial(tests[i][0]) == tests[i][1]);
    }
}
```

Summary

- Test Driven Development
- Refactoring

References

- The Agile Samurai by Jonathan Rasmusson, The Pragmatic Programmers, 2010